B Defining FUN in K

FUN is a functional language. It is overall much simpler than most existing functional languages, though it has an interesting combination of functional language features, such as: different parameter passing styles, such as call-by-value, call-by-name, and call-by-need; functions defined via currying as well as tuple arguments (however, tuples are not values, they can only be used in function definitions and function invocations); let and letrec bindings; lists together with basic operations on them; referencing, dereferencing and side effects. The following are valid FUN programs:

```
-----
P1:
-----
   letrec
      max{val} l{val} (x{val}, y{val})
        = if (* x) != y
          then -1
          else if null?(cdr l)
               then (car l)
               else let x{val} = max (cdr l) ((x := (* x) + 1 ; x), y + 1)
                    in if (x <= car l)
                       then (car l)
                       else x
    and
      map{val} f{val} l{val} = if null? l
                               then list()
                               else cons (f (car l)) (map f (cdr l))
    and
      factorial{val} x{val} = if x <= 0
                              then 1
                              else x * factorial(x - 1)
    in max (map factorial list(1, 2, 3, 4, 5, factorial 5)) (ref 1, 1)

------
P2:
------
    let f{name} = let c{val} = ref 0
                  in (
                       c := (* c) + 100 ;
                       fun x{need} -> (
                                         c := (* c) + 1000 ;
                                         x + x + (* c)
                                       )
                     )
    in let y{val} = ref 0
       in f(y := (* y) + 1 ; * y) + f(0)
```

Once the semantics of FUN is completed (see Appendix B.2), we can also execute programs like the ones above. With our Maude implementation of the K definition of FUN running on a 1.5GHz Windows Tablet PC, we got the following results in about 5 seconds:

```
result P1:  668950291344912705758811805409037258675274633313802981029567135 2301
            633557244962989366874165271984981308157637893214090552534408589 4081
            218598984811143896500059649605212569600000000000000000000000000000
result P2:  2202
```

Changing f{name} by f{need} in the second program would produce the answer 3302.

B.1 Defining TRIVIA-FUN

Before we define the actual FUN language, whose definition is rather intricate, let us first define a simpler call-by-value variant, with functions taking only one argument and with bindings binding only one name, with no references and with no lists:

K-Annotated Syntax of TRIVIA-FUN

$$
\begin{array}{rcll}
\mathit{Nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \text{ all natural numbers} & \\
\mathit{Int} & ::= & \ldots \text{ all integer numbers} & \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} & \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} & \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Name} & \\
 & \mid & \mathit{Exp} + \mathit{Exp} & [\mathit{strict},\ \mathit{extends} +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \ldots & & \\
 & \mid & \mathsf{fun}\ \mathit{Exp}\ \texttt{->}\ \mathit{Exp} & \\
 & \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mathsf{let}\ \mathit{Name} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & \\
 & \mid & \mathsf{letrec}\ \mathit{Name} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} & [\mathit{strict}(1)]
\end{array}
$$

K Configuration and Semantics of TRIVIA-FUN

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{closure}(\mathit{Name}, K, \mathit{Env}) \\
\mathit{KResult} & ::= & \mathit{Val} \\
\mathit{Env} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Loc}] \\
\mathit{Store} & ::= & \mathit{Map}[\mathit{Loc}, \mathit{Val}] \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{env}(\mathit{Env}) \mid \mathit{store}(\mathit{Store}) \mid \mathit{nextLoc}(\mathit{Loc}) \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![K]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] \\
K & ::= & \ldots \mid \mathit{restore}(\mathit{Env}) \mid \mathit{bindTo}(\mathit{List}[\mathit{Name}]) \mid \mathit{writeTo}(\mathit{List}[\mathit{Name}])
\end{array}
$$

$[\![e]\!] = [\![k(e)\ \mathit{env}(\cdot)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(\mathit{loc}(0))]\!]$
$[\![\langle k(v)\rangle]\!] = v$

$$
\begin{array}{l}
k(\underline{\quad x \quad}\rangle\ \mathit{env}(\rho)\ \mathit{store}(\sigma) \\
\quad \sigma[\rho[x]] \\
k(\cdots\cdots\ \mathsf{let}\ x = e\ \mathsf{in}\ e' \cdots\cdots\rangle\ \mathit{env}(\rho) \\
\quad e \curvearrowright \mathit{bindTo}(x) \curvearrowright e' \curvearrowright \mathit{restore}(\rho) \\
k(\cdots\cdots\cdots\ \mathsf{letrec}\ x = e\ \mathsf{in}\ e' \cdots\cdots\cdots\rangle\ \mathit{env}(\rho) \\
\quad \mathit{bindTo}(x) \curvearrowright e \curvearrowright \mathit{writeTo}(x) \curvearrowright e' \curvearrowright \mathit{restore}(\rho) \\
k(\cdots\ \mathsf{fun}\ x\ \texttt{->}\ e\ \cdots\rangle\ \mathit{env}(\rho) \\
\quad \mathit{closure}(x, e, \rho) \\
k(\cdots\cdots\ \mathit{closure}(x, e, \rho)\ v \cdots\cdots\rangle\ \mathit{env}(\underline{\rho'}) \\
\quad v \curvearrowright \mathit{bindTo}(x) \curvearrowright e \curvearrowright \mathit{restore}(\rho') \qquad\qquad \rho \\
\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to e_1 \\
\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to e_2
\end{array}
$$

The definitions of the auxilliary computation items *restore*, *bindTo* and *writeTo* are so useful in language definitions, that, in our Maude implementation of K, they are part of the prelude file which is included in all K definitions. These and many others can be found in Appendix D.

B.2 Untyped FUN

K-Annotated Syntax of Untyped FUN

Nat	::=	$0 \mid 1 \mid 2 \mid \ldots$ all natural numbers	
Int	::=	$\ldots$ all integer numbers	
Bool	::=	true | false	
Name	::=	all identifiers; to be used as names of variables and functions	
Exp	::=	$Int \mid Bool \mid Name$	
	|	$Exp + Exp$	$[strict,\ extends\ +_{Int\times Int\to Int}]$
	|	$Exp - Exp$	$[strict,\ extends\ -_{Int\times Int\to Int}]$
	|	$Exp * Exp$	$[strict,\ extends\ *_{Int\times Int\to Int}]$
	|	$Exp\,/\,Exp$	$[strict,\ extends\ quotient_{Int\times Int\to Int}]$
	|	$Exp\,\%\,Exp$	$[strict,\ extends\ remainder_{Int\times Int\to Int}]$
	|	$-Exp$	$[strict,\ extends\ -_{Int\to Int}]$
	|	$Exp < Exp$	$[strict,\ extends\ <_{Int\times Int\to Bool}]$
	|	$Exp <= Exp$	$[strict,\ extends\ \leq_{Int\times Int\to Bool}]$
	|	$Exp > Exp$	$[strict,\ extends\ >_{Int\times Int\to Bool}]$
	|	$Exp >= Exp$	$[strict,\ extends\ \geq_{Int\times Int\to Bool}]$
	|	$Exp == Exp$	$[strict,\ extends\ =_{Int\times Int\to Bool}]$
	|	$Exp\ \texttt{!=}\ Exp$	$[strict,\ extends\ \neq_{Int\times Int\to Bool}]$
	|	Exp and Exp	$[strict,\ extends\ \wedge_{Bool\times Bool\to Bool}]$
	|	Exp or Exp	$[strict,\ extends\ \vee_{Bool\times Bool\to Bool}]$
	|	not Exp	$[strict,\ extends\ \neg_{Bool\to Bool}]$
	|	fun $ParamListSeq$ -> Exp	[fun $pls\ pl$ -> e = fun pls -> fun pl -> e]
	|	$Exp\ ExpList$	$[strict(1),\ f(el_1, e, el_2) \rightleftharpoons e \curvearrowright f(el_1, \Box, el_2)]$
	|	let $Binding$ in Exp	
	|	letrec $Binding$ in Exp	
	|	if Exp then Exp else Exp	$[strict(1)]$
	|	ref Exp	$[strict]$
	|	$*Exp$	$[strict]$
	|	$Exp := Exp$	$[strict]$
	|	$\&Name$	
	|	list E	$[\mathsf{list}(el_1, e, el_2) \rightleftharpoons e \curvearrowright \mathsf{list}(el_1, \Box, el_2)]$
	|	car Exp	$[strict]$
	|	cdr Exp	$[strict]$
	|	null? Exp	$[strict]$
	|	cons Exp Exp	$[strict]$
	|	Exp ; Exp	$[strict]$
ExpList	::=	$List^{()}_{\_,\_}[Exp]$	
Param	::=	$Name\{ParamPassStyle\}$	
ParamPassStyle	::=	val | name | need	
ParamList	::=	$List^{()}_{\_,\_}[Param]$	
ParamListSeq	::=	$ParamListSeq\ ParamList$	
Binding	::=	$ParamListSeq = ExpList$	$[(pls\ pl = e) = (pls = \text{fun}\ pl\ \text{->}\ e)]$
	|	$List_{\_\text{and}\_}[Binding]$	$[(pl = el\ \text{and}\ pl' = el') = (pl, pl' = el, el')]$

K Configuration and Semantics of Untyped FUN

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Loc} \mid \mathit{unit} \\
& \mid & \mathit{closure}(\mathit{List}[\mathit{Name}], K, \mathit{Env}) \mid \mathit{frozen}(K, \mathit{Env}) \mid \mathit{unfreeze}(K, \mathit{Env}) \mid \mathsf{list}(\mathit{List}[\mathit{Val}]) \\
\mathit{KResult} & ::= & \mathit{Val} \\
\mathit{KProper} & ::= & \ddagger \mathit{List}[K] \ddagger \mathit{List}[K] \ddagger \mathit{List}[\mathit{Name}] \ddagger \mathit{List}[K] \ddagger \\
& \mid & \mathit{mkFrozenVal}(K) \mid \mathit{mkUnfreezeVal}(K) \\
\mathit{KLabel} & ::= & @\mathit{let} \mid @\mathit{letrec} \mid @\mathit{closure} \\
\mathit{Env} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Loc}] \\
\mathit{Store} & ::= & \mathit{Map}[\mathit{Loc}, \mathit{Val}] \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{env}(\mathit{Env}) \mid \mathit{store}(\mathit{Store}) \mid \mathit{nextLoc}(\mathit{Loc}) \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![K]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] \\
K & ::= & \ldots \mid \mathit{restore}(\mathit{Env}) \mid \mathit{bindTo}(\mathit{List}[\mathit{Name}]) \mid \mathit{writeTo}(\mathit{List}[\mathit{Name}])
\end{array}
$$

$[\![e]\!] = [\![k(e)\ \mathit{env}(\cdot)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(\mathit{loc}(0))]\!]$

$[\![\langle k(v) \rangle]\!] = v$

functions & application

$k(\underline{\langle\ \mathsf{fun}\ pl\ \texttt{->}\ e\ \rangle}\ \mathit{env}(\rho)$
$\quad \overline{\mathit{closure}(pl, e, \rho)}$

$\mathit{closure}(pl, e, \rho)\ el = \ddagger pl \ddagger el \ddagger \cdot \ddagger \curvearrowright @\mathit{closure}(e, \mathit{restore}(\rho))$

$k(\underline{\langle\ \ddagger \cdot \ddagger \cdot \ddagger xl \ddagger el \ddagger \curvearrowright @\mathit{closure}(e, k)\ \rangle}\ \mathit{env}(\rho)$
$\quad \overline{\mathit{strict}(el) \curvearrowright k \curvearrowright \mathit{bindTo}(xl) \curvearrowright e \curvearrowright \mathit{restore}(\rho)}$

let

$\mathsf{let}\ pl = el\ \mathsf{in}\ e = \ddagger pl \ddagger el \ddagger \cdot \ddagger \curvearrowright @\mathit{let}(e)$

$k(\underline{\langle\ \ddagger \cdot \ddagger \cdot \ddagger xl \ddagger el \ddagger \curvearrowright @\mathit{let}(e)\ \rangle}\ \mathit{env}(\rho)$
$\quad \overline{\mathit{strict}(el) \curvearrowright \mathit{bindTo}(xl) \curvearrowright e \curvearrowright \mathit{restore}(\rho)}$

letrec

$\mathsf{letrec}\ pl = el\ \mathsf{in}\ e = \ddagger pl \ddagger el \ddagger \cdot \ddagger \curvearrowright @\mathit{letrec}(e)$

$k(\underline{\langle\ \ddagger \cdot \ddagger \cdot \ddagger xl \ddagger el \ddagger \curvearrowright @\mathit{letrec}(e)\ \rangle}\ \mathit{env}(\rho)$
$\quad \overline{\mathit{bindTo}(xl) \curvearrowright \mathit{strict}(el) \curvearrowright \mathit{writeTo}(xl) \curvearrowright e \curvearrowright \mathit{restore}(\rho)}$

call-by-value

$\ddagger(\underset{\cdot}{x\{\mathsf{val}\}}\rangle \ddagger(\underset{\cdot}{e}\rangle \ddagger\langle \underset{x}{\cdot}\ \rangle \ddagger\langle \underset{e}{\cdot}\rangle \ddagger$

$k(\underline{\langle\ x\ \rangle}\ \mathit{env}(\rho)\ \mathit{store}(\sigma)$ when $\sigma[x]$ is not of the form $\mathit{frozen}(e, \rho')$ or $\mathit{unfreeze}(e, \rho')$
$\quad \overline{\sigma[\rho[x]]}$

call-by-name

$\ddagger(\underset{\cdot}{x\{\mathsf{name}\}}\rangle \ddagger(\underset{\cdot}{e}\rangle \ddagger\langle \underset{x}{\cdot}\ \rangle \ddagger\langle \underset{\mathit{mkFrozenVal}(e)}{\cdot}\ \rangle \ddagger$

$k(\underline{\mathit{mkFrozenVal}(e)\rangle}\ \mathit{env}(\rho)$
$\quad \overline{\mathit{frozen}(e, \rho)}$

$k(\underline{\langle\ x\ \rangle}\ \mathit{env}(\underline{\rho}{\scriptstyle \rho'})\ \mathit{store}(\sigma)$ when $\sigma[x]$ is $\mathit{frozen}(e, \rho')$
$\quad \overline{e \curvearrowright \mathit{restore}(\rho)}$

call-by-need

$\ddagger(\underset{\cdot}{x\{\mathsf{need}\}}\rangle \ddagger(\underset{\cdot}{e}\rangle \ddagger\langle \underset{x}{\cdot}\ \rangle \ddagger\langle \underset{\mathit{mkUnfreezeVal}(e)}{\cdot}\ \rangle \ddagger$

$k(\underline{\mathit{mkUnfreezeVal}(e)\rangle}\ \mathit{env}(\rho)$
$\quad \overline{\mathit{unfreeze}(e, \rho)}$

$k(\underline{\langle\ x\ \rangle}\ \mathit{env}(\underline{\rho}{\scriptstyle \rho'})\ \mathit{store}(\sigma)$ when $\sigma[x]$ is $\mathit{unfreeze}(e, \rho')$
$\quad \overline{e \curvearrowright \mathit{writeTo}\,\rho[x]\ \mathit{andKeep} \curvearrowright \mathit{restore}(\rho)}$

references

$k(\underline{\mathsf{ref}\ v\rangle}\ \mathit{store}(\underline{\quad \sigma \quad})\ \mathit{nextLoc}(\underline{\quad l \quad})$
$\quad \overline{l} \qquad\qquad \overline{\sigma[l \leftarrow v]} \qquad\qquad \overline{l +_{\mathit{Int}} 1}$

$k(\underline{\&\ x\rangle}\ \mathit{env}(\rho)$
$\quad \overline{\rho[x]}$

$k(\underline{*\ l\ \rangle}\ \mathit{store}(\sigma)$
$\quad \overline{\sigma[l]}$

$k(\underline{l := v\rangle}\ \mathit{store}(\underline{\quad \sigma \quad})$
$\quad \overline{\mathit{unit}} \qquad\qquad \overline{\sigma[l \leftarrow v]}$

$\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_1, \quad \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_2$

$\mathsf{car\ list}(v, vl) \rightarrow v, \quad \mathsf{cdr\ list}(v, vl) \rightarrow \mathsf{list}(vl), \quad \mathsf{null?\ list}() \rightarrow \mathsf{true}, \quad \mathsf{null?\ list}(v, vl) \rightarrow \mathsf{false}, \quad \mathsf{cons}\ v\ \mathsf{list}(vl) \rightarrow \mathsf{list}(v, vl)$

$\mathit{unit}; v \rightarrow v$